CHAPTER 2
CLOSE TO THE WIRE

This chapter studies PC host requests on the USB bus from the bottom up:

- We look at the signaling on the wires and discover an underlying real time structure of frames and microframes.
- Within these (micro) frames is a packetized signaling scheme at one of three standard speeds.
- We learn that sequences of packets are used to generate transactions.
- We list the special packets that a PC host uses to communicate with hubs that support different speed devices.
- We learn that control transactions are used during enumeration and that three additional transactions types (interrupt, bulk and isochronous) are used during run time
- We study the available control requests that a PC host can use to interact with an I/O device and a hub device.

DIFFERENTIAL SIGNALING

If you were to put an oscilloscope on the USB data wires, you would see a pair of differential signals at one of three standard speeds. The two data wires, D+ and D-, are driven at the same time and generally in anti-phase (There are a few single-ended signaling methods and these are covered later). The USB data wires are connected point-to-point and the signaling is half-duplex, which means that only one end of the wire pair is driven at a time. The protocol, presented below, defines how the two ends of the wire pair take turns transmitting information. In the following discussion, the upstream device is the **transmitter,** and the downstream device is the **receiver** unless noted.

The USB data wires do not include a CLOCK signal, so the communication between nodes is described as asynchronous. The base speed of each pair of USB data wires is negotiated during enumeration (described in Chapter 3) and a SYNC signal is transmitted prior to the data transfer so that the receiver can tune its nominal bus clock to the exact transitions of the transmitter. When receiving a signal from the bus, a device will typically oversample the signals so that transitions can be better detected. Later we'll see that a CLOCK signal is embedded in the data.

The IDLE condition of a high speed bus in D+ low and D- low. The IDLE condition of a full speed bus is D+ high and D- low. The IDLE condition of a low speed bus is D+ low and D- high. The operation at all speeds is the same so, for the sake of clarity, only one set of diagrams will be shown. The IDLE state will be called the J state while the active state will be called the K state – this allows one set of diagrams to apply to all three bus speeds. The time scale of the diagrams is dependent upon the base speed of each pair of USB data wires.

THE FUNDAMENTAL PACKET

The fundamental element of communication on the USB data bus is a **packet**. A packet consists of three pieces: a start, some information, and an end, as shown in Figure 2-1.

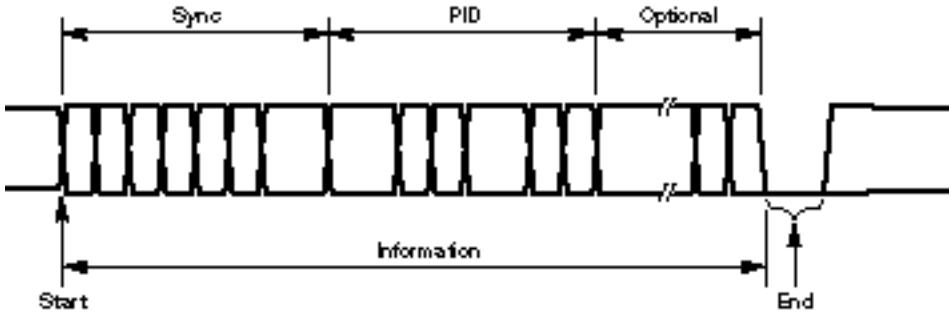


Figure 2-1. Anatomy of a packet

The start of a packet is signaled by a transition out of the J state into the K state. More transitions are driven by the transmitter on the next available times, to produce a SYNC sequence. The receiver uses this SYNC sequence to tune its receive clock with the transitions of the received data, thus ensuring reliable reception of the information portion of the packet. The SYNC sequence terminates with two K states and the packet information starts in the next bit-time.

The SYNC pattern generated by the PC host for a high speed bus will contain 32 bits of SYNC (KJKJK....KJKK). Some of these SYNC bits will be consumed by intervening hubs but the farthest end device is guaranteed to see at least 12 bits of SYNC – this will be sufficient to lock the receive clock. Full speed and low speed devices use 8 bits of SYNC so data will start in bit-time eight.

The packet information varies from one byte to 1025 bytes. The first byte is always a Packet Identifier, or PID, that will define how the other information bytes should be interpreted. A packet identifier byte is formed with 4 bits and the complement of these 4 bits; this redundancy allows the receiver to error-check the PID. The PID encoding is shown in Table 2-1.

Table 2-1. USB packet types

PID Value	Packet Type	Packet Category	PC Host-Hub only	High Speed Only
0101	SOF	Token		
1101	SETUP	Token		
1001	IN	Token		
0001	OUT	Token		
0011	DATA0	Data		
1011	DATA1	Data	x	
0111	DATA2	Data	x	iso only
1111	MDATA	Data		iso only
0010	ACK	Handshake		
1010	NAK	Handshake		
1110	STALL	Handshake		
0110	NYET	Handshake		x
1100	PRE	Special	x	
1100	ERR	Special	x	x
1000	SPLIT	Special	x	x
0100	PING	Special		x
0000	RESERVED	RESERVED		

There are four categories of packets. **Token** packets are used to set up **Data** packets, which are acknowledged by **Handshake** packets. There are also **Special** packets used to implement “speed-conversion” connections. The next section discusses each packet type in detail. Each will be presented as a discrete building block, and we’ll use a sequence of these building blocks to define a robust communications channel.

The last part of a packet is an End-Of-Packet identifier. A high speed EOP pattern is 40 bit-times without a transition – this will cause a bit-stuff error (see below) which is expected. A full and low speed connection drives both D+ and

D– low for two bit times to signal this End-Of-Packet. This is not a differential signal. It is an easily identified single-ended zero, or SE0.

As an aside, it is worth mentioning the term “bit stuffing” that is used in the USB specification. The data lines on the bus reflect the information that is being transmitted. There is no separate CLOCK signal, and the receiver relies on regular signal transitions to maintain synchronism with the transmitter. A nonreturn to zero inverted, or NRZI, transmission encoding is used on the data lines. The encoding and decoding occur at the transmitter and receiver, respectively, so the process is transparent to all other parts of USB. The process will, however, be observed on the bus.

The NRZI protocol requires the following:

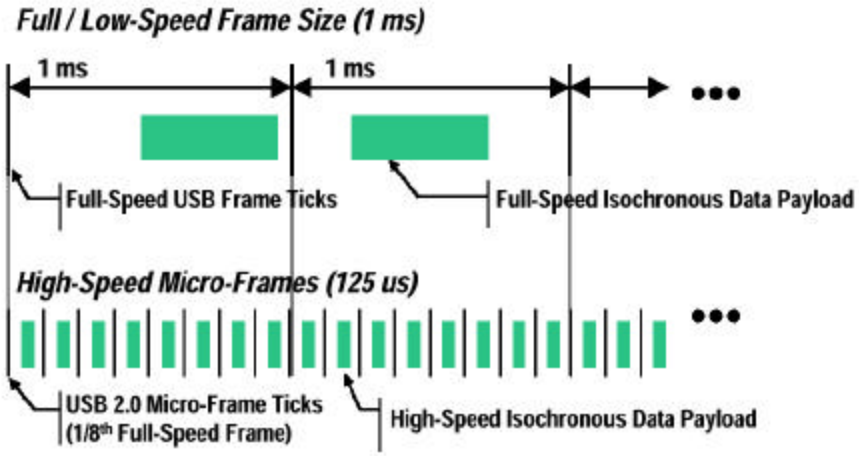
- Toggle each bit time for multiple data 0s.
- Do not toggle each bit time for multiple data 1s.
- Toggle each data 1 to 0 pair.
- Do not toggle each data 0 to 1 pair.

This scheme results in no bus transitions for long sequences of data 1s, and as a result, the receiver could lose synchronism. The USB specification requires that a 0 data bit is added, or stuffed, after six consecutive 1 data bits to ensure that the bus transitions often enough. We do not have to account for this during any part of the design, but if you put an oscilloscope on the bus, you will see these “extra” bits.

DIFFERENT PACKET TYPES

Start-Of-Frame Token Packet

The root hub transmits a SOF packet every 1.0 milliseconds. The time between two SOF packets is called a **frame**. A high-speed root hub will also transmit eight microframes as shown in Figure 2-2, using additional SOF packets. The addition of microframes does add some control complexity to a high speed link but it saves a great deal of buffer complexity. Without microframes a high speed device would need to be able to buffer (1.00/480-MHz = 480,000 theoretical bit times or 1Mbyte of data between frames! Microframes and the addition of DATA2 and MDATA packet types allows high speed devices to be designed with 40KB buffers. <<I thought it was 3K. My math says 40K. What did I do wrong?>> A high speed link has a practical upper limit of about xxMB/sec.



<< Figure 8-14 from USB 2.0 Spec. >>

Figure 2-2. USB signaling scheme uses frames and microframes

For a full speed link there are (1.00 ms/12 MHz) = 12,000 theoretical bit times or 1500 bytes of data, maximum within a frame. The SYNC and EOP overhead will reduce this number to a practical upper limit of 1200 bytes of data that could be transmitted in each frame. A low speed link has a practical upper limit of yy bytes per frame. The different speeds allow you to select one that best matches your application.

The USB signaling scheme includes robust error checking. This Cyclic Redundancy Check, or CRC, code is generated by the transmitter and received by the receiver. The receiver also generates a CRC code over the same data bits and compares this with the received code. The PID is not included in the CRC calculation. If they are different, the packet is rejected and not acted upon.

A SOF packet has 11 bits of data and 5 bits of CRC error checking (Figure 2-3).

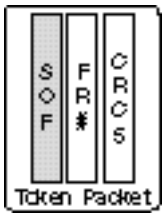


Figure 2-3. Start-Of-Frame packet

The 11 bits of data in a packet is a monotonically increasing frame number. It has no absolute value but can be used by real-time devices to synchronize their data transfer. This number rolls over every 2048 milliseconds, or approximately every 2 seconds.

The eight microframes within a 1msec frame time all have the same frame number. If high-speed devices need to synchronize to a microframe granularity then they should count SOF tokens once the frame number changes.

So with no other USB activity on the bus, our oscilloscope would see a SOF packet every 125 us on high-speed bus and every 1.0 msec on a full speed bus. A low speed bus does not see a SOF token – refer to the “Special Packets” section later in the chapter. Any device attached to the bus could receive this SOF packet and use it as a regular “heartbeat” signal. We shall see in later chapters that real-time devices, such as audio and video, rely heavily on this regular SOF packet.

The SOF packet is the only packet that does not have a destination address. It is broadcast for all USB devices to use and does not require an acknowledgment.

IN, OUT and Setup Token Packets

The other three token packets, IN, OUT, and SETUP, have the same format as shown in Figure 2-4. They contain a 7-bit device address, a 4-bit endpoint address, and a 5-bit CRC. The 7-bit device address will specify one of 126 possible device addresses (0 is reserved, the root hub takes up one address) and the 4-bit endpoint address is a sub-address within the device.

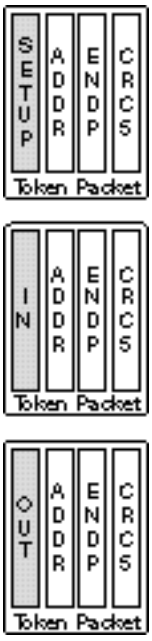


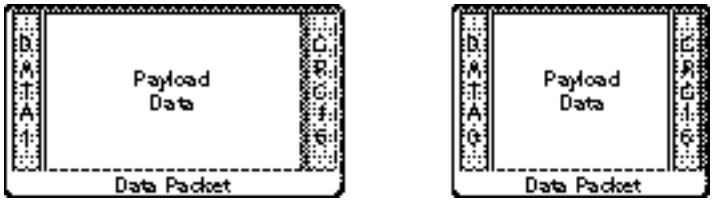
Figure 2-4. IN, OUT and Setup token packets

These three token packets are used to set up data transfers between the PC host and a specific data source or sink on a specific device (this data source or sink is called an **endpoint**; it is defined in the next chapter).

- An IN packet sets up a data transfer from the device to the PC host, and an OUT packet sets up a data transfer from the PC host to the device.
- IN and OUT packets can address any endpoint on any device.
- A SETUP packet is a special case of an OUT packet; it is "high priority," which means that the device MUST accept it even if this means aborting a previous command. A setup packet is always targeted at the bidirectional control endpoint 0.

Data Transfer Packets

The data transfers initiated by the SETUP, IN, and OUT token packets are implemented with DATA0, DATA1, DATA2 and MDATA packets (Figure 2-5).



<< Add Data2 and Mdata >>

Figure 2-5. Data transfer packets

A data transfer packet can have a payload varying from 0 to 1023 bytes and a 16-bit CRC. DATA2 and MDATA packets are only used for high-speed isochronous data transfer – this special case is covered later in this chapter. There are multiple data packet types to provide error detection. A transmitter will typically alternate DATA0 and DATA1 packets, and the receiver should check that alternate DATA0 and DATA1 packets are being received. For example, if two DATA0 packets are received back-to-back, this indicates a loss of a DATA1 packet and is an error condition.

Handshake Packets

Handshake packets are used by a receiver to indicate the good, bad, poor or ugly reception of token and/or data packets. Figure 2-6 shows the four handshake packet types: ACK, NAK, NYET and STALL.

A handshake packet consists only of PID. There is no CRC; the redundant encoding in the PID provides the required error-checking.

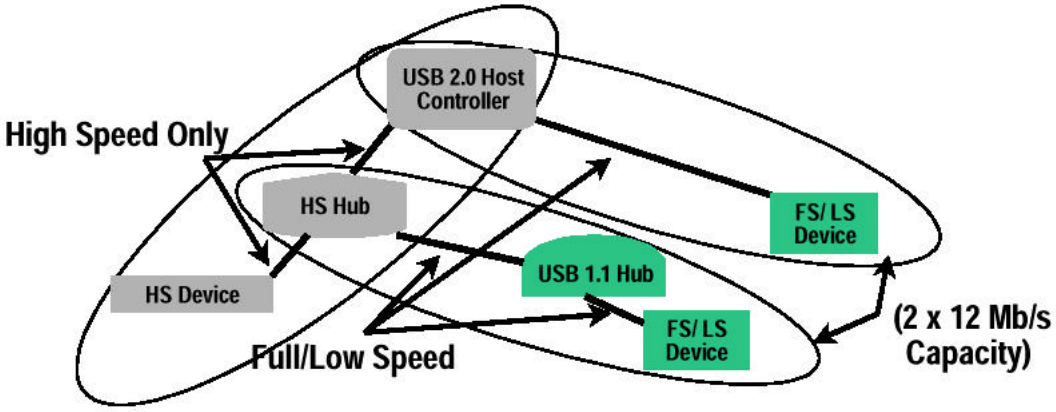
- An ACK handshake indicates the successful reception of a token and/or data packet.
- A NAK handshake indicates that the receiver is currently too busy or doesn't have the resources to deal with the token and/or data packet right now. A device is allowed to NAK all transactions, except a SETUP token, while the PC host is not allowed to NAK any transactions (it should always have the time and resources to receive a packet; this will simplify the buffering on an I/O device).
- High-speed devices support an improved NAK mechanism using a NYET handshake. NAKing an OUT transaction is not efficient since the data has already been transmitted on the bus. This results in poor bus utilization if there is a high frequency of NAKed out transmissions. A high-speed device may use a PING special token (described in the next section) to inquire if the receiver has buffer space to accept an OUT transaction - if an ACK is returned then the transmitter will schedule an OUT, if a NAK is returned then the transmitter will continue to inquire with PINGs. This results in better bus utilization.
- If something is very wrong at the device, then a STALL handshake is used to tell the PC host that some help is required. For example, an I/O device will generate a STALL handshake for all commands that it doesn't understand.

<< new "horizontal" diagram >>

Figure 2-6. Handshake packets

Special Packets

Most of the special packets are used by the PC host to communicate with hubs that have to manage a change in bus speed for an I/O device. As such, they are not a concern for an I/O device designer. Figure 2-7 shows a typical USB configuration with three different speed connections identified.



<< This is Figure 5-6 from USB 2.0 Spec. – derive a better one >>

Figure 2-7. Typical USB network supports all speeds

The PC host will communicate at high-speed to a high-speed hub but consider the case shown in Figure 2-7 where a full speed device is attached to this high-speed hub. The high-speed hub will "store and forward" data that is exchanged between the high-speed PC host and the full-speed device. The PC host will use SPLIT transactions to minimize data buffering in the hub but these tokens will not be visible to the full-speed device. Only the basic Data Transfer Packets (DATA0 and DATA1) will be seen on the full-speed link. An additional special token, ERR, is required to deal with errors that can occur inside a SPLIT transaction. A full discussion of split transactions is beyond the scope of this I/O device design book and the interested reader is referred to the USB 2.0 specification, Chapter 8 and 11 for a full discussion.

The root hub knows the operating speed of each USB device, so the root hub knows when it is about to initiate a transaction with a full or low-speed device (how the speed is known is described in the next chapter). A high-speed, root hub will pass packets destined for full and low-speed devices to the high-speed hub that is supporting these full/low speed devices. This high-speed hub will "store" the packet and "forward" it, at the correct speed, towards the addressed device. The hub prefixes a special PRE token before all other tokens it generates for a low-speed device to warn other downstream hubs that a low-speed transaction is coming. Note that a low-speed device can handle interrupt and control transactions; it does not support bulk or isochronous transactions.

Figure 2-8 shows a representative low-speed interrupt transaction. I won't describe this in detail because the I/O device itself does not see these PRE tokens and therefore does not need to deal with them. The hub acts on the PRE tokens by passing them through to all full-speed downstream ports and passing only the following token to low-speed downstream ports.

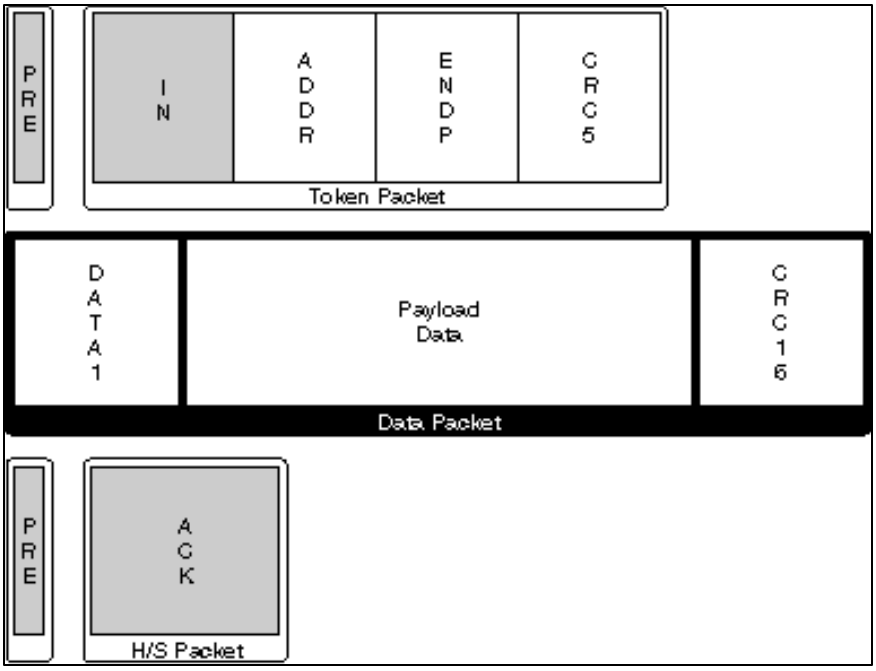


Figure 2-8. Low-speed packets are eight times longer

BUILDING A TRANSACTION

A predefined sequence of packets, called a transaction, is used to define data movement between the PC host and an I/O device. Each transaction **must** occur within the same (micro) frame; a transaction is not allowed to straddle multiple (micro) frames. Once a sequence is started, it must be completed with no intervening packets from other transactions.

The USB specification defines four different transaction types to handle different types of data that will be transmitted over the bus. All the transactions use the same building-block packets just presented but differ in how the packets are scheduled and the response to an error. The PC host software has two parameters to deal with—delivery **time** accuracy and delivery **quality** accuracy. The time and quality attributes have different importance to different data types. Table 2-2 summarizes the four transaction types and highlights the delivery attributes of each type.

Table 2-2. Four transaction types

Type	Important Attributes	Maximum Size	Example
Interrupt	Quality + Time	64 (8 for LS)	Mouse, keyboard
Bulk	Quality	64	Printer, scanner
Isochronous	Time	1023	Speakers, video
Control	Quality + Time	64 (8 for LS)	System control

<< ADD a column for High speed and tabulate values>>

The PC host guarantees time accuracy by reserving portions of a frame for isochronous and control transfers.

The PC host guarantees quality accuracy by using a handshake mechanism. In general:

- If data is received correctly, then an ACK handshake is generated.
- If there is a problem with the data transfer, a NAK handshake is generated.
- If the data receiver is confused, a STALL handshake is generated.

The next section looks at each transaction type individually and studies their characteristics.

In the following diagrams, which depict the different types of transactions, the packets transmitted by the PC host are shown in white and the packets transmitted by the I/O device are shown in black.

Interrupt Transactions

Using the term “Interrupt Transaction” for this implementation is very generous! Hardware designers think of an Interrupt as an immediate response to an external event. But this is not how USB Interrupt Transactions occur. Instead, the PC host **polls** the I/O device to inquire if it needs attention. The PC host can poll as often as every (micro) frame, but for many mechanical or human-related devices, polling once every 10 frames, or 100 times a second, will be an ample response time. If your full-speed I/O device needs attention faster than 1 millisecond, then the microcontroller should preprocess the data and handle a response locally, and a status update should be sent to the PC host at a convenient time later.

Figure 2-9 shows several interrupt transfers.

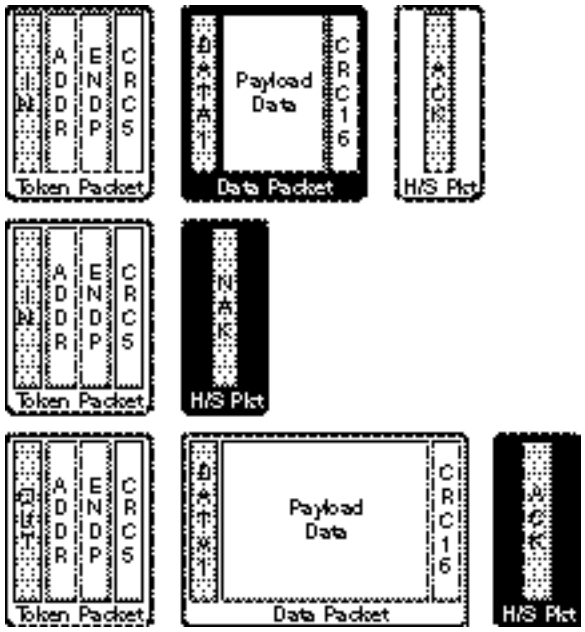


Figure 2-9. Interrupt transfers

The first sequence in Figure 2.9 shows a successful data transfer from an I/O device to the PC host. The data payload size varies from 0 to 64 bytes (0 to 8 for a low-speed device). The ACK is a positive response for a valid data transfer. To be able to respond so quickly, the I/O device must have had the data ready and be waiting for an IN token. If the I/O device did not have data ready, it would generate a NAK as shown in Figure 2-9 (center), and the root hub would retry later.

The third sequence shows a successful data transfer from the PC host to an I/O device. Note that interrupt OUT transactions were added in USB Specification V1.1. In this example, the I/O device must have had a buffer ready to accept the data, because it was able to respond immediately.

Interrupt transactions are efficient: if the I/O device has no new data since the previous time it was polled, it can respond with a NAK. This preserves bus bandwidth, and the transaction will not be retried until the next scheduled polling interval.

If the I/O device is confused, it responds with a STALL, and the PC host alerts higher levels of software so that the situation at the I/O device can be resolved.

Bulk Transactions

The packets used to implement bulk data transactions are identical to those used for interrupt transactions—the difference is the system scheduling. USB provides "good delivery" of bulk data. The transaction quality is guaranteed such that no data is lost, but the transaction time is not guaranteed. Bulk transactions use the time available in a (micro) frame after all of the guaranteed transactions have been scheduled. This means that a long printout to a USB printer will take longer on a busy USB bus. On the other hand, if there is a lot of time available within a (micro) frame for bulk transactions, the PC host can schedule multiple transactions to the same I/O device in a single (micro) frame. The data packets will alternate between DATA0 and DATA1 tokens for error checking.

An I/O device can NAK an OUT data packet if, for example, the device has no buffer space to put the incoming data in. The PC host will retry the transaction later. A high-speed device can use the PING-NYET protocol to improve bus utilization if a high frequency of NAKs is anticipated.

Isochronous Transactions

The packets used to implement isochronous data transactions are similar to those used for bulk transactions but have different system scheduling and are not acknowledged. Before a PC host agrees to support isochronous data transactions to/from an I/O device, the PC host negotiates a guaranteed data delivery schedule. Isochronous transfers occur **every** (micro) frame, and the PC host will ensure there is available bandwidth within the frame before agreeing to set up the connection. Once set up, the I/O device is guaranteed a slice of every (micro) frame; however, the position within the frame is not guaranteed, so the I/O device must buffer the data and allow for delivery time jitter.

Figure 2-10 shows some full or low-speed isochronous packets; there is no handshake packet with each data transfer. Isochronous packets are time-dependent, and the late delivery of a packet is as useless as no delivery, so bad packets are not retried. An I/O device will typically use the data from a prior packet again. DATA0 tokens are used with a data payload that varies from 0 to 1023 bytes.

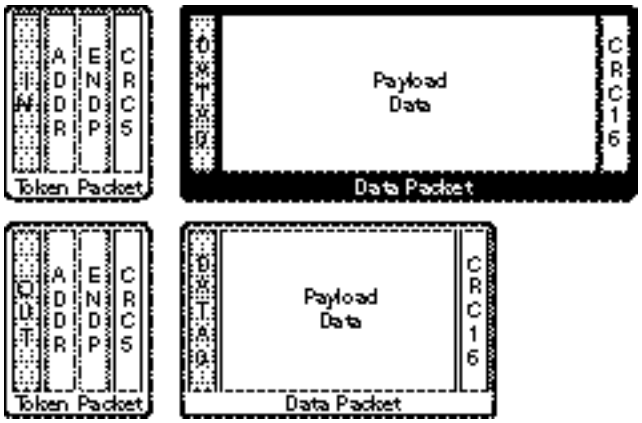
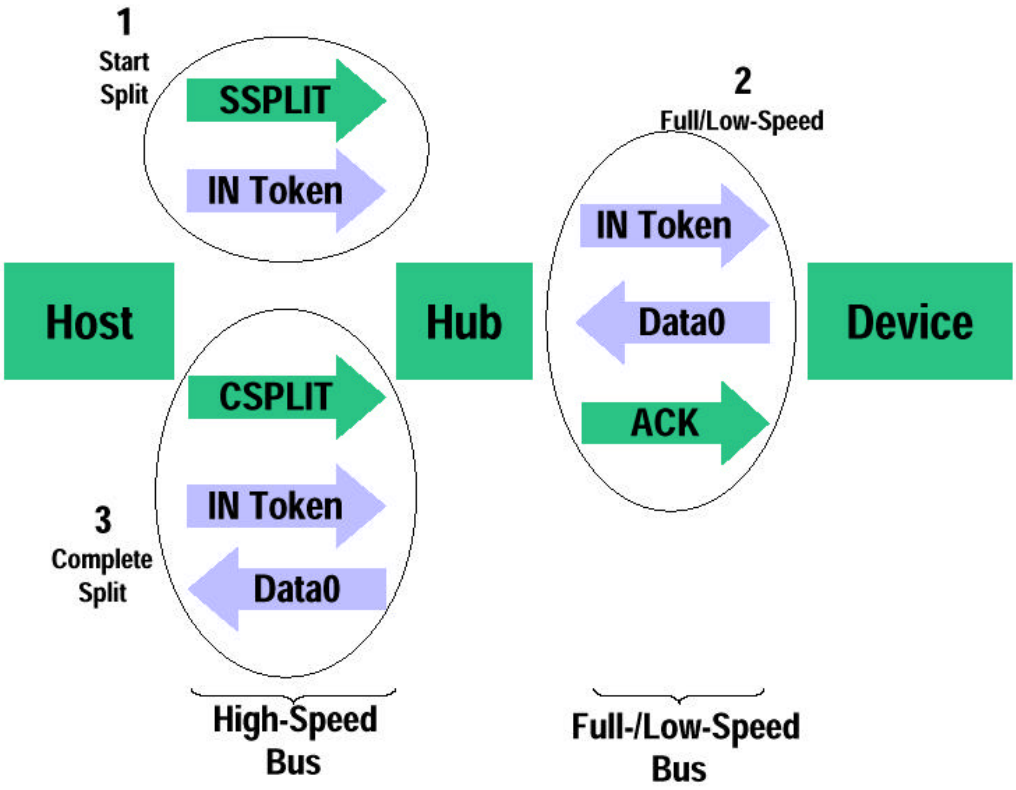


Figure 2-10. Full or low-speed isochronous packets

For very high bandwidth full-speed devices up to three isochronous transactions can be scheduled providing a bandwidth of ??MB/s. Additional tokens are used to provide error checking for the three transaction read and three transaction write cases shown in Figure 2.11.



<< fixup this diagram from USB 2.0 spec (Fig 8-8) >>

Figure 2-11. Very high bandwidth high-speed isochronous transactions

Control Transactions

Control transactions are the most complicated. Control transactions need a lot of system protocol overhead to ensure that the commands and data are correctly received. This is not a major concern for system performance, because these transfers typically occur only during enumeration or command initialization. A control transactions is divided into three phases, each of which uses some of the building-block packets already defined. These phases, shown in Figure 2-11, always start with a setup phase and end with a status phase. The data transfer phase is optional. All control transfers address endpoint 0 on the targeted I/O device. Each phase may be in a different (micro) frame, separated by many other (micro) frames.

<< "Horizontal" version of same figure from Book 1 >>

Figure 2-11. Control transfer phases

The setup phase consists of a setup packet, a DATA0 packet, and a handshake packet (Figure 2-11). The DATA0 packet always contain 8 bytes, and the format of this data is predefined (and presented in the next section). The handshake packet is always an ACK, because the I/O device is not allowed to NAK or STALL a setup packet. A setup packet must always be accepted even if this requires aborting the execution of a previous control transfer request.

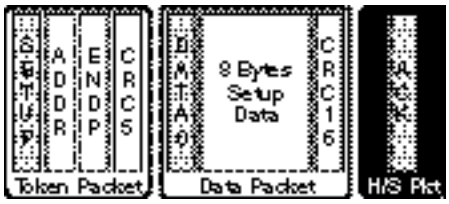


Figure 2-12. Setup phase of a control transfer

The setup phase will specify if a Data Phase is required. Some setup commands can be completely specified by the 8 bytes of data in the setup phase, and others require more data to be written to, or read from, the I/O device. Some setup commands require a lot of data to be read from the I/O device. If there is more data than can be contained in a single packet, then the I/O device must send the data in multiple packets; Figure 2-13 shows a multipacket response. The maximum size of a packet is implementation-dependent but must be a minimum of 8 bytes (64 for high-speed). Other legal sizes for a full-speed device are 16, 32, and 64. The next chapter discusses how a particular I/O device informs the PC host of the maximum-sized packets that will used. Figure 2-13 shows a typical control read data from the I/O device. This phase consists of three transactions that could take place in three different (micro) frames. The first two data packets are the largest size that the I/O device can provide while the last packet is a "short" packet. The end of the data phase is signaled by this short packet. If the data payload fit into an exact multiple of "largest size" packets then the I/O device must provide a zero-length short packet to signal the end of the data phase.

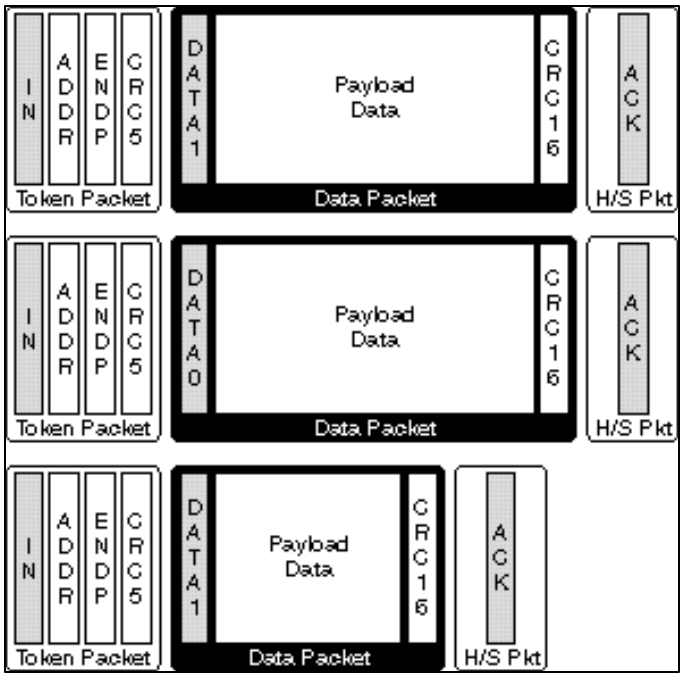
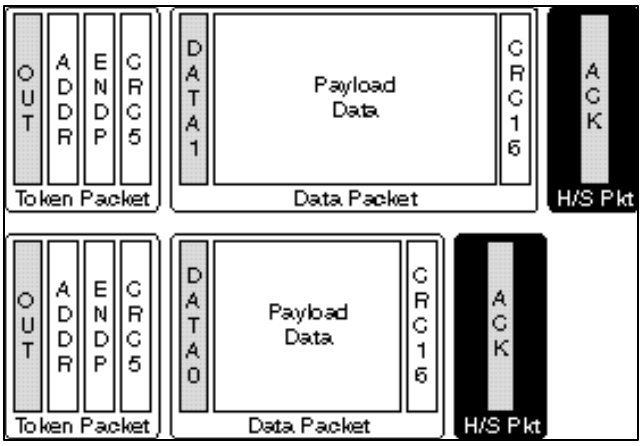


Figure 2-13. Control read data phase

The PC host may want to provide more data to the I/O device—this would be specified in the setup phase. A control write data phase is used in this case. The example in Figure 2-14 uses multiple packets to supply all of the data. The "short packet" protocol is not required for a PC host write since the exact size of the transfer is specified in the setup phase. In the example shown in Figure 2-14 the I/O device was not fast enough to process the data sent by the PC host in the first DATA1 transaction so it NAKed the DATA0 transaction. The PC host retried the transfer later, probably waiting several (micro) frames, and it was accepted by the I/O device at the second attempt.



<< Add a NAKed packet to this diagram >>

Figure 2-14. Control write data phase

A control transaction always ends with a status phase (Figure 2-15):

- If there is no data phase, then the I/O device is acknowledging receipt of the setup phase.
- If the data phase is a control read, then the root hub is acknowledging receipt of all the data from the I/O device.
- If the data phase is a control write, then the I/O device is acknowledging receipt of all the data.

The status phase is an IN transaction if the I/O device is providing the status or an OUT transaction if the PC host is providing the status. A zero-length data packet is used to signify success, and a NAK or STALL response denotes an error condition.

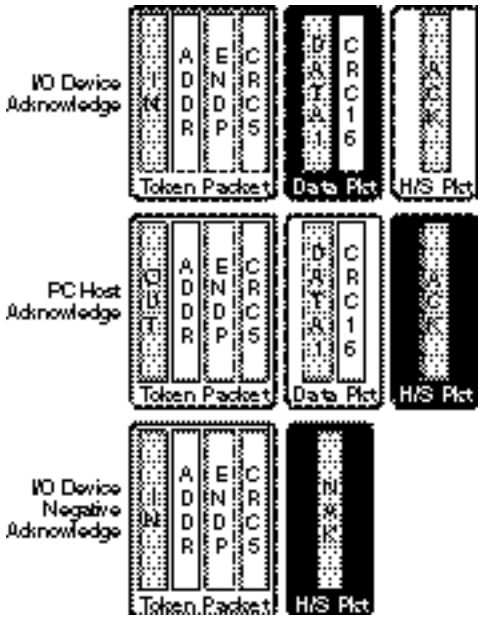
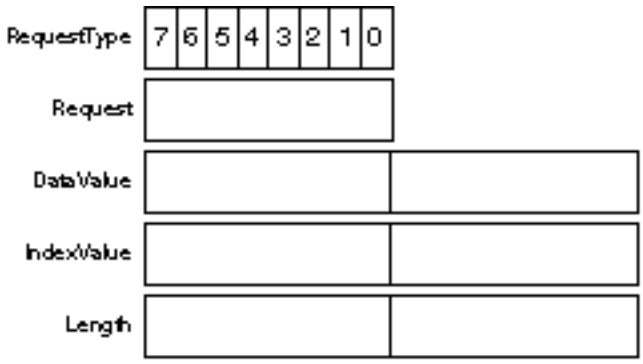


Figure 2-15. Final status phase of a control transfer

PC HOST REQUESTS

Figure 2-16 shows a fixed format for the 8-byte DATA0 packet within a control transaction.



Definition of Request Type Bit Fields

Bit 7	0 = Transfer Host Data to Device		1 = Transfer Device Data to Host	
Bit 6-5	00 = Standard	01 = Class	10 = Vendor	11 = Reserved
Bit 4-0	0000 = Device All other codes are Reserved	00001 = Interface	00010 = Endpoint	00011 = Other

Figure 2-16. Format of a PC host request

The **RequestType** parameter is a bit field with bit 7 indicating the direction of the transfer for the next transaction. Bits 6 and 5 are a top-level switch of how the **Request** byte at offset 1 should be interpreted—this chapter covers only Standard Requests and Class Requests for a hub device. Other class requests are covered in later chapters. The lower four bits specify the destination of this request; the PC host can send requests to the device, to an interface, or to an endpoint.

The Request parameter indicates which information the PC host wants. Table 2-3 lists the standard requests for a device along with the required action by the I/O device. Table 2-4 is a similar list for a hub device.

Table 2-3. Standard PC host requests of an I/O device

Request	Required Device Action
Get_Status	Return current status
Clear_Feature	Clear Specified Feature
Set_Feature	Set Specified Feature
Set_Address	Store Unique USB Address and use from now on
Get_Descriptor	Return requested descriptor
Set_Descriptor	Set Specified Descriptor
Get_Configuration	Return Current Configuration or 0 if not configured
Set_Configuration	Set Configuration to the one specified
Get_Interface	Return Current Interface
Set_Interface	Set Interface to the one specified
Sync_Frame	Synchronize USB Frame Numbers (Async device)

The remaining 6 bytes of the PC host Request are used as data to support the request. If a single byte or word **Data Value** is required, it is supplied in byte offset 2 and/or 3. If a single **Index Value** is required, it is supplied in byte offset 4 and/or 5. The **Length** word in byte offset 6 and 7, if greater than 1, specifies the total length of a subsequent data transfer.

The optional second phase of a control transfer moves any required data to support the request between the PC host and the I/O device.

It is easier to understand these requests and their response through an example. In the next chapter we'll step through an enumeration sequence and identify all requests and responses.

Table 2-4. PC host requests of a hub device

Type	Request	Required Device Action
Standard	Get_Status	Return current status
Standard	Clear_Feature	Clear Specified Feature
Standard	Set_Feature	Set Specified Feature
Standard	Set_Address	Store Unique USB Address and use from now on
Standard	Get_Descriptor	Return requested descriptor
Standard	Set_Descriptor	Optional: Set Specified Descriptor
Standard	Get_Configuration	Return Current Configuration or 0 if not configured
Standard	Set_Configuration	Set Configuration to the one specified
Standard	Get_Interface	Optional: Return Current Interface
Standard	Set_Interface	Optional: Set Interface to the one specified
Standard	Sync_Frame	Optional: Synchronize USB Frame Numbers
HUB Class	Get_Bus_State	Optional (for diagnostics): Return D+, D–
HUB Class	Get_Hub_Status	Return Hub Status with Changed Identified
HUB Class	Get_Hub_Descriptor	Return Hub Descriptor
HUB Class	Set_Hub_Descriptor	Optional: Set Hub Descriptor
HUB Class	Set_Hub_Feature	Enable a standard hub feature
HUB Class	Clear_Hub_Feature	Disable a standard hub feature
HUB Class	Get_Port_Status	Return Port Status with Changed Identified
HUB Class	Set_Port_Feature	Enable a standard port feature
HUB Class	Clear_Port_Feature	Disable a standard port feature

ERROR HANDLING

A lot of care and effort went into creating a robust communications channel for USB. I have briefly mentioned some of the factors during this "Close to the wire" discussion. If we were building a USB transceiver and Serial Interface Engine, then I would go into much more detail at this point. The focus of this book, however, is **using** USB. There are many USB transceivers available (see a listing on the CD-ROM), and all of the implementations have been verified for correctness to the USB specification. If you are a silicon designer who needs to include a transceiver on your ASIC, please refer to the reference design material in the Chapter 2/Silicon Design directory on the CD-ROM and then contact the USB Implementors Forum (www.usb.org) and request information on the "USB MacroCell Library".

CHAPTER SUMMARY

This chapter provided insight into the signals on the bus, the fundamental packetized nature of the bus, and the transactions used to exchange data on the bus. The PC host uses a defined set of requests to control all of the devices attached to the bus, and these devices need to respond in a defined manner. It is interesting and educational to use an oscilloscope to look at the differential signals on the USB data wires, but this is not a productive tool to use in debugging a system. What we need is an observation tool that understands the packetized nature of the bus and can display the bus transactions at a higher level; a variety of these will be presented in Chapter 5.